# Appendix A

## Interlisp-VAX Study Report

Addendum to
Interlisp-VAX: A Report

STAN-CS-81-879
(HPP-81-14)

Dave Dyer
University of Southern California
Information Sciences Institute
Marina del Rey, California

January 20, 1982

Since Larry Masinter's "Interlisp-VAX: A Report" is being circulated widely, it is important that it be as accurate as possible. This note represents the viewpoint of the implementors of Interlisp-VAX, as of January, 1982.

The review of the project and the discussions with other LISP implementors that provided the basis for the Report took place in June, 1981.

We believed at the time, and still believe now, that the Masinter Report is largely a fair and accurate presentation of Interlisp-VAX, and of the long-term efforts necessary to support it. We now have the advantage of an additional 6 month's development effort. There are some areas where progress and performance have been better than anticipated in the Report, and we would like to report on our current status.

### AVAILABILITY AND FUNCTIONALITY

Interlisp-VAX has been in use for testing purposes both here at ISI and at several sites around the ARPANET, since November, 1981.

We are planning the first general release for February, 1982--ahead of the schedule that was in effect in June, 1981.

The current implementation includes all of the features of Interlisp-10 with very few exceptions. There is no noticeable gap in functionality among Interlisp-10, Interlisp-D and Interlisp-VAX, except for features that are inherently peculiar to some implementations (e.g., windows on the Dolphin, JSYS and TENEX on the PDP-10).

Among the Interlisp systems we are running here are KLONE, AP3, HEARSAY, and AFFIRM.

PERFORMANCE

Masinter's analysis of the problems of maximizing performance was excellent, both for Interlisp generally and for the VAX particularly. It is now reasonable to quantify the performance based on experience with real systems. The analysis of the performance of Lisp programs is quite complex, and single numbers are often more misleading than representative. It is hard to give a complete analysis, so we will only give general performance numbers.

CPU speed (on a single-user VAX/780) is, for many of the programs we have measured, currently in the range of 1/4 the speed of Interlisp-10 (on a single-user DEC 2060). We believe that a factor of two overall performance improvement is achievable.

Currently, it seems reasonable to allow 1 MByte real memory per active user.

# Interlisp-VAX: A Report

Larry Masinter
Xerox Palo Alto Research Center
Palo Alto, California

August 1, 1981

Contents:



The views expressed in this report are those of the author; they do not necessarily reflect those of the Xerox Corporation, Stanford University, or the University of Southern California.

This study was funded in part through the SUMEX Computer Project at Stanford University under grant RR-00785 from the Biotechnology Resources Program of the National Institutes of Health.

## I. INTRODUCTION

Since November 1979, a group at the Information Science Institute of the University of Southern California has been working on an implementation of Interlisp for the DEC VAX-series(1) computers. This report is a description of the current status, future prospects, and estimated character of that Interlisp-VAX implementation. It is the result of several days of discussion with those at ISI involved with the implementation (Dave Dyer, Hans Koomen, Ray Bates, Dan Lynch); with John L. White of MIT, who is working on an implementation of another Lisp for the VAX (NIL); with the implementors of Interlisp-Jericho at BBN (Alice Hartley, Norton Greenfeld, Martin Yonke, John Vittal, Frank Zdybel, Jeff Gibbons, Daryle Lewis); with the implementors of Franz Lisp and Berkeley Unix(2) at U.C. Berkeley (Richard Fateman, Bill Joy, Keith Sklower, John Foderaro); and with my colleagues at Xerox PARC.

An earlier draft of this report was circulated to the parties involved in the Interlisp-VAX discussions. This document has been revised as a result of comments received.

Why Interlisp-VAX? In early April, 1981, a meeting was held at SRI of ARPA-sponsored or related Lisp users, to discuss the status and future of Lisp. Those of the community who were current Interlisp users felt strongly that: (1) there was a need for Interlisp to continue to be a viable programming environment in the 1980's, strongly standardized among all implementations; and (2) the most important new implementation of

Interlisp would be for the VAX. There were several reasons for the choice of both the VAX and Interlisp.

Why VAX? The primary reason is that many sites already have VAXes, where they are and will continue to be used not only for Lisp and AI research, but also for use as general purpose, time-shared computing resources, for running FORTRAN, business computing, etc. The VAX is considered to be the most important "technology transfer" vehicle for Interlisp AI programs in the early through mid '80s: it is already spread widely throughout industry and industrial laboratories; and it is very widespread among ARPA's military clientele. It is unlikely that researchers who develop application-oriented AI systems in Interlisp will want to re-implement them in some other language, and it is unlikely that these institutions (private and military) will buy machines specifically for AI programs if those programs constitute only an occasional part of their computing needs. The VAX is believed to be the most likely vehicle for transferring applications to those institutions.

In addition, VAX (for better or for worse) appears to be the machine that many computer science departments around the country have chosen for their "next generation" machine. Insofar as there is a need to spread the concepts and software technologies developed in Interlisp to these departments, it is believed that there is a need to have Interlisp running on the VAX.

Why Interlisp? The Interlisp programming environment has been in wide use in the Artificial Intelligence community for a number of years. It is a powerful, integrated environment, having evolved over the years into a stable system. The availability of multiple, compatible implementations on a number of machines means that researchers can easily transport their programs from any implementation to another.

Why this report? Because of the perceived importance of Interlisp-VAX to the community, and because of my experience with Interlisp and its implementations, I was asked by Stanford and ISI to evaluate the status of the on-going project at ISI, and to estimate the magnitude of the tasks remaining, expected performance and character of the resulting product. Many ongoing research institutions are making plans for their future computational requirements, and many of the decisions about choice of programming language and hardware hinge on the prospects for Interlisp-VAX. In light of the large amount of confusion in the community about the future availability of Interlisp on a VAX, it was thought important to have an outside assessment of the future of the project.

## II. INTERLISP-VAX: OVERVIEW AND STATUS

### A. Project definition and history

While the Interlisp-VAX project started in November 1979, most of the first year was taken up with project startup and training of personnel (none of the project members were originally familiar with the C programming language, UNIX, or Interlisp, either as a programming

environment or its overall implementation). Thus, most of the work on the implementation to date has been accomplished since November 1980.

The goal of the project has been to produce a version of Interlisp which runs on a VAX, which:

Is as compatible with Interlisp-10 as practical.
While it is difficult to give a metric for compatibility, the goal of the project is that most Interlisp programs in the community will run in Interlisp-VAX merely by recompiling them. At a bare minimum, the Interlisp-VAX code must run the standard Interlisp packages such as the Interlisp Editor, Masterscope, Break Package, Record Package, DWIM, CLisp, and so forth.

Uses the extended virtual address space of the VAX.
One of the primary motivations for investing in the VAX is that the VAX potentially has a large (at least 2**30 bit) address space. Many Interlisp-10 users long ago have run out of address space, and spend much of their time trying to squeeze programs into available address space.

Has adequate performance.
The Lisp produced is expected to make reasonable use of the hardware. It is difficult to give a single number which describes the performance of a system (because some things will run faster and some slower), but the average performance of Interlisp-VAX must be within a factor of 2 of other Lisps which run on the VAX (e.g., Franz and NIL). In addition, Interlisp-VAX must be competitive in price/performance to a DEC-20 for the size of programs which it is now able to run, and also be able to handle larger programs.

B. Summary of the Interlisp-VAX architecture

Interlisp-VAX is a non-microcoded implementation more similar in architecture to Interlisp-10 than Interlisp-D or Interlisp-Jericho. This is appropriate for the VAX, which has a powerful "native" instruction set and is time-shared between a number of users, not all of whom would be running Interlisp. Interlisp-VAX is intended to run on top of the Berkeley Unix operating system. Unlike Interlisp-10, in which the kernel is written in assembly language, the kernel of Interlisp-VAX is written in the high-level systems implementation language C. This might well simplify the transportation of Interlisp-VAX to another machine which had a C compiler and similar characteristics (byte addressable memory, UNIX, 32-bit registers).

Without going into great detail, the important aspects of the Interlisp-VAX architecture are as follows: deep binding; full implementation of "spaghetti stacks"; compilation to VAX native code (with no "block compiler"); memory allocation in 64 KByte "sectors" with sector-table giving type per sector; no CDR-coding (CONS cells take 64 bits); a "stop and copy" garbage collector; 31-bit immediate integers (with plans, but no implementation, of "bignums"). These design choices seem reasonable for the VAX, with exceptions noted below in the section on performance.

Interlisp-VAX has the following component pieces:

1. Machine-independent "higher-level" Interlisp code. This includes, for example, the Interlisp editor, file package, the Masterscope program analyzer. This code is shared, intact, with Interlisp-10, Interlisp-D and Interlisp-Jericho.

2. Interlisp-D code. This is Lisp code which, although shared with Interlisp-D (and Interlisp-Jericho), is not used in Interlisp-10. For example, the implementation of Terminal Tables and Read Tables may be shared with the other Interlisp implementations.

3. VAX-specific Lisp code. This code is necessary to interface to the C kernel and perform other VAX-specific operations. For example, the implementation of the DATATYPE package, while in Lisp, must satisfy constraints placed by the Interlisp-VAX garbage collector, and thus has some essential differences from the version of the DATATYPE package for other Interlisp implementations.

4 C kernel. The C kernel handles memory management, garbage collection, the interpreter, "Spaghetti stack" support (including FUNARGs, RESUME for processes/coroutine support), bootstrapping and interface to the operating system.

5. Lisp/C interface. A small amount of VAX machine code is necessary for the interface between Lisp and code generated by the C compiler. Primarily of interest here is the code which is part of function call and return.

6. VAX code-generator. The VAX native-code generator takes the output of the Interlisp-D Byte-Compiler and generates VAX native code. The Byte-Compiler [Masinter & Deutsch 1980] is a machine independent optimizing compiler which produces intermediate "linearized lisp" code for an abstract stack machine.

C. Current implementation status of Interlisp-VAX: what's been done?

Many of the major design decisions for Interlisp-VAX have been made, including layout of memory, important code sequences (e.g., function call and return for all of the various cases), representations of pointers and system data types, and many parts of the interface to the operating system. In addition, the following tasks have been accomplished:

1. Higher level Interlisp software. The "shared" Interlisp software has been examined, and a few problems identified and fixed; the rest will run in Interlisp-VAX with little change.

2. Interlisp-D code. An initial pass over the Interlisp-D code identifying which portions can be shared has been made.

3. VAX-specific Lisp code. The major pieces which have been written are a version of the DATATYPE package, an array package, and the compiled code loader and parser.

4. C kernel. Most of the C kernel has been completed, in the sense that the code is there and has passed preliminary tests.

5. Lisp/C interface. This has been completed.

6. VAX code-generator. A first version of the VAX code generator has been produced and, to a great extent, debugged. The important design decisions about function call sequences, as well as some of the important open-coding sequences (e.g., CAR and CDR), have been made.

D. Tasks remaining in existing code

1. Higher level Interlisp software. Problems may arise in implementing Interlisp's notions of files, versions, and dates under UNIX; if so, it may be necessary to fix those portions of the Interlisp higher-level software to be more implementation independent.

2. Interlisp-D code. Unfortunately, Interlisp-D is a "moving target" and it is difficult to rely on the sources staying compatible. Insofar as code is shared between Interlisp-D and Interlisp-Jericho, the same code will most likely run under Interlisp-VAX. Problems may arise insofar as the lower levels of Interlisp-VAX differ.

3. Vax-specific Lisp code. The DATATYPE implementation requires some work. The array package seems to be relatively complete, although the program has not been extensively tested. The compiled code loader/parser has been completed and tested in "cross-compilation" mode, while running in Interlisp-10.

   The VAX/UNIX I/O package still requires much work. The interface between Interlisp and UNIX is to be accomplished via (1) the Interlisp-D FILEIO package, which gives an interface to buffered, random access files from higher level Interlisp software, (2) some VAX-specific Lisp code, which then interfaces to (3) some pieces of the C kernel. The interfaces between many of these pieces are being designed, but some of the pieces have not been written.

   There is a body of the Interlisp environment which, although nominally not part of the "core" of Interlisp, forms a useful part of most of its implementations. For example, the DIRECTORY package and GETFILEINFO are Interlisp-10 facilities which, while not part of the Moore VM document, can be implemented in Interlisp-VAX, are part of Interlisp-10, Interlisp-D and Interlisp-Jericho, and are used by Interlisp application programs.

Interface with UNIX's notion of terminals and interrupts has been considered, but the final details have not been worked out. Initial versions of Interlisp-VAX will have a very simple notion of interrupts.

4. C kernel. Future changes will likely be required depending on the needs of the Lisp-level I/O package, interrupts, and a new version of UNIX which will allow Interlisp to use the high end of memory. The C kernel contains some especially "tricky" areas: interpreter, stack management and garbage collection. These were not completed as of June 1981. Experience with other Lisp/Interlisp implementations has been that debugging and complete testing are difficult. Bugs often are found in the handling of obscure and rare cases, as the code interacts with many other parts of the system. I expect Interlisp-VAX to have its share of problems in these areas.

5. Lisp/C interface. Changing the Lisp/C interface will only be necessary in response to fixing some of the expected "performance bugs" of Interlisp-VAX, e.g., free-variable-pointer-caching (discussed below) may require changes in the function-call sequences.

6. VAX code-generator. My examination of the Vax code generator uncovered a few minor problems due to a misunderstanding of conventions required by the ByteCompiler, and undoubtedly a few more will surface.

   More importantly, the current code generator for VAX native code will (as planned) require much work to bring it to the point where it generates production-quality code. In particular:

   a. A register-allocating version of the code generator (in some ways a complete rewrite) would significantly improve performance on the VAX.

   b. A "peephole optimizer" for VAX instructions would enable Interlisp-VAX to take advantage of the VAX's complex instruction repertoire.

   c. More "open" compilation of frequently used routines will be necessary in many circumstances. Although many open-coding sequences have been incorporated, adding more will of course require additional time and effort.

   d. Modification of the ByteCompiler to suppress boxing of intermediate results would have payoff in speed for integer calculations and space for floating arithmetic.

E. Other areas requiring work

In addition to the areas outlined above, a number of other areas need attention:

1. Free variable pointer caching. There is a very serious performance problem in Interlisp-VAX, the correction of which will require major changes to the Interlisp-VAX system. Interlisp-VAX uses deep binding. While deep binding is a reasonable choice for Interlisp-D (because of microcoded free variable lookup) it may be a source of a large performance penalty in Interlisp-VAX, especially in interpreted code. In any case, there is currently no mechanism for "caching" free variable pointers, and so free variables are "looked up" at every reference, even within an inner loop. This is clearly unacceptable. A design needs to be worked out and integrated into the compiler and stack access mechanism. No one scheme is clearly optimal. It is clear that whatever scheme is chosen will require changes to the compiler, interpreter, garbage collector and stack manipulation routines.

2. Bootstrapping. Bootstrapping is as complicated in Interlisp-VAX as it is in other Interlisp implementations for a variety of reasons. For example, debugging "low level" pieces of the system is made more difficult because bootstrap-load order requirements are difficult to detect without running the (time consuming) bootstrap process. This traditionally is merely a source of frustration rather than an insurmountable barrier.

3. Documentation. Documentation of Interlisp-VAX discussing its differences from other Interlisps and areas such as interface to UNIX is needed. In addition, there is some intention to participate in the upcoming major revision of the Interlisp Reference Manual.

4. Access to UNIX facilities. Interface from Lisp to UNIX facilities such as pipes, processes, Shell programs will greatly increase the utility of Interlisp-VAX. These facilities are not necessary for running current Interlisp-10 programs except insofar as they replace Interlisp-10 facilities (e.g., SUBSYS).

5. SYSOUT. The current Interlisp-VAX SYSOUT facility dumps the entire allocated virtual memory of the Lisp system (currently, without any of the "shared" Interlisp code over 1 MByte.) At some future date, Berkeley UNIX will provide a mechanism which will allow writing out individual pages and a page map, making SYSOUT files more manageable.

6. Porting to other VAX operating systems. Many sites do not run the Berkeley UNIX operating system, instead choosing VMS (the DEC-supplied operating system for the VAX), or EUNICE (a UNIX compatibility package developed at SRI.) These are candidates for

"other implementations" of Interlisp-VAX. Because of Interlisp's heavy use of the operating system's memory management facilities, porting Interlisp-VAX to these other operating systems will likely prove quite difficult.

## III. WHAT WILL INTERLISP-VAX BE LIKE?

Assuming the above tasks are completed, the question remains: what will it be like? There are two issues: in what way will Interlisp-VAX differ from other Lisp implementations, and what performance can be expected?

### A. Comparison of Interlisp-VAX to other systems

Full Interlisp-VAX is intended to be highly compatible with Interlisp-10, to the point where many complex programs would move gracefully between it and other Interlisp implementations; the only areas of incompatibility are those which are necessarily not shared between any implementations: access to machine code within Lisp routines, etc. In addition, there are currently no plans for "linked" function calls in Interlisp-VAX, nor for a "block" compiler. These are minor difficulties.

Interlisp-VAX will be able to access some of the facilities of the UNIX environment to good effect, e.g., one might imagine using it as an interactive "shell" programming language.

Interlisp-VAX will not have any particular CAPABILITIES for bit-mapped graphics.

Interlisp-VAX will have a larger "small" arithmetic range.

### B. Performance

There are two major factors in the performance of Interlisp on the VAX: the first is in the actual CPU time to complete various operations, and the second is in the amount of time spent paging.

#### 1. CPU performance

The performance profile of a Lisp system is complex, and there are many areas where Interlisp-VAX's relative performance to other Interlisp implementations will vary over a wide range. There seem to be a few areas of critical performance to any program: function call, variable reference, data structure access, arithmetic, and garbage collection. An appropriate weighted average of performance in those areas is a good overall measure of total system performance.

One important way of estimating performance of Interlisp-VAX is to use the code in other Lisp implementations for the same task as a comparison, taking into account the differences in the various code sequences. Comparisons are made between Interlisp-VAX and Franz, NIL, and Interlisp-10.

a. Function call and return

A function call for Interlisp-VAX will be at least twice as slow as a similar function call in Franz Lisp, partly because of language requirements (Franz does not check that the number of arguments passed matches the number of arguments expected) and partly because of the design of the Interlisp-VAX stack format (variable names are pushed as well as the values.)

In Franz Lisp, a minimal call/return takes 17 microseconds (VAX 11/780). Call/return in Interlisp-VAX may be as high as 100 microseconds, although the average will most likely be nearer to 40 microseconds.

In Interlisp-10 on a DEC 2060, a block-internal call takes on the order of a microsecond (PUSHJ, POPJ), the minimal (non-block) call/return takes 57 instructions (roughly 25 microseconds) while some functions, because of the Interlisp Swapper, may take more than 200 instructions for call/return (100 microseconds). The variation in function call time will apparently be high for Interlisp-VAX and Interlisp-10; for some functions, Interlisp-VAX function call will be slightly faster; for calls which in Interlisp-10 would be block internal, an Interlisp-VAX call might be 50 times slower. Note that benchmarks which purport to make comparisons with Interlisp-10 should explicitly control for the possibly enormous variation in Interlisp-10 function call time.

b. Variable reference

Performance ON LOCAL variable reference in Franz and Interlisp-VAX will be similar if Interlisp-VAX delivers its optimizing, register allocation code generator. Currently, variable reference will often be slightly slower. More importantly, free variable access will be very significantly slower in Interlisp-VAX, even after a variable caching scheme is implemented, because of the cost of variable lookup when using deep binding.

c. Garbage collection

The "stop and copy" variety of garbage collection, while compacting the address space and thus reducing the working set of subsequent computations, is more expensive in CPU time and memory usage than the "mark and sweep" variety, by a nominal factor of two. Garbage collections, even using mark and sweep, for large address space systems can be expensive. A full VAXSYMA garbage collection is reported to take on the order of 3 seconds cpu time. A garbage collection of a 2 MByte address space in Rutgers Lisp [Hedrick] on a 2060 with extended virtual addressing reportedly took 20 seconds cpu time. These figures are not particularly

consistent. It seems likely that (1) garbage collection is swap limited, and (2) the respective operating systems used to gather those times do not do a particularly good job of filtering out swap overhead from cpu time. It is not unreasonable to expect, however, that a Interlisp-VAX garbage collection will take twice as long as a Franz Lisp collection, because of the intrinsic overhead of "stop and copy" over "mark and sweep".

An alternative computation can be made as follows: Assuming an Interlisp-VAX system to use 4 MBytes of memory, then with a compacting garbage collection but no other memory localization algorithms, I believe that most user programs would "dirty" at least 1/4 of all system pages (i.e., 1 MByte) within a relatively small amount of time. Let us suppose a garbage collection occurs after a user has allocated the equivalent 40K CONS cells, or .32 MB of storage. This would involve referencing 1.6 MB of memory. This would mean that a garbage collection would take, at a minimum, between 2 and 20 seconds of CPU time on a VAX 11/780.

## 2. Paging Performance and Real Memory Requirements

I spent a considerable amount of time trying to estimate the number of users or sizes of Lisp systems that some typical VAX configurations might support. I believe that this is one of the most important factors in Interlisp-VAX performance, because of the predicted large virtual address spaces of Interlisp-VAX programs (one of the main reasons for going to Interlisp-VAX in the first place.)

### a. Operating system considerations

Interlisp-VAX will be implemented on top of the Berkeley UNIX operating system. Another possible candidate for a host operating system is a UNIX compatibility package by the name of EUNICE, written at SRI, which runs under DEC-supplied operating system VMS. There is some controversy over the relative performance and functionality of VMS vs. UNIX. A fairly comprehensive set of benchmarks [Kashtan] showed that VMS out-performed UNIX in a variety of paging configurations. It is claimed by the Berkeley UNIX implementors that (a) many of the benchmarks were atypical of real computations, and (b) tests were run on an early version of Berkeley UNIX, and performance has improved considerably since then. I believe that the choice of operating system can be made on grounds other than predicted performance for running Interlisp: reliability, maintenance, cost, etc. and further, that converting Interlisp-VAX to run under EUNICE rather than Berkeley UNIX will be a relatively minor job compared to the magnitude of the Interlisp-VAX implementation itself. It seems that the difference between operating systems makes for only a relatively small factor in the overall performance, if the real memory available is too small to hold the "working set" of the programs attempting to run at any one time.

### b. Real memory requirements of Interlisp-VAX

There are a variety of ways of estimating memory needs. The best estimates seem to come from: (1) comparison with MACSYMA in Franz Lisp (VAXSYMA), (2) comparison with Interlisp-10 and Interlisp-D.

1) Virtual Address space (minimum). Many current Interlisp-10 programs run with a virtual address space of 2 MByte (2 full "forks"). A similar system, in Interlisp-VAX, will probably require 4 MByte of address space because: (1) there is expansion for 32 rather than 18 bit addresses (no CDR coding); (2) the copying garbage collector will require, when it runs, twice the allocated space; and (3) Interlisp-VAX allocates storage in quanta of 64 KByte sectors rather than a 2 KByte "page" as in Interlisp-10, giving more "breakage" per datatype. This figure is consistent with numbers extrapolated from Interlisp-D.

2) Working set. In current Interlisp-10, the "working set" of many programs is .5 MByte or more (that is, the amount of real memory outside of the "system" necessary to keep the program from spending more than half of its time paging). Extrapolating using the same figures as above, the working set of a "typical" Interlisp-VAX application will be over 1 MByte. This figure is consistent with memory requirements extrapolated from Interlisp-D.

3) Calculation of real memory requirements. If there are i users, j of whom are active, they will need i*31 KBytes of page table (31 KByte = 4 MByte/128), plus .75*j MByte bytes for their working set. For example, 5 users, 2 of whom are actively running at any one time, would require less than 2 MByte of real memory (outside of i/o buffers, etc.).

However, if systems increase in allocated space (independent of the working set) because more programming or data is contained in their virtual address space, one might imagine a situation where the virtual address spaces were in the 20-30 MByte range. (Many users do not believe that a 2**24 byte virtual address space, 16 MByte, is big enough for applications they plan in the near future.) In such a situation, each such Interlisp process would require as much as .2 MByte of real memory for its page table, independent of its activity. This might severely limit the number of users who could be active on the system at any one time.

c. Problem areas

There are some problem areas, both with UNIX and with VMS, which will have to be resolved:

1) Sharing. VMS currently has more flexibility in allowing sharing of space among users in a piecemeal fashion. In the current Interlisp-VAX design, only .1 MBytes of the address space are "pure" in the sense that Berkeley UNIX would allow it to be shared among multiple users. Insofar as multiple users have the same large virtual address space (e.g., they are running the same program with a large, fairly static "knowledge base"), sharing is important to improving the number of users allowable at any one time.

2) Problems with large virtual address space. VMS requires disk/swap space to be pre-allocated, at system generation time, for the maximum allowable in the system. With multiple users with large address space programs, this adds considerably to the amount of disk space required on

the system (even if most of those users are inactive.) In addition, VMS requires an additional swap file to be pre-allocated, which contains J*W pages, where J is the maximum number of processes with independent address spaces (100 would not be an unreasonable figure for a machine used by many users for editing, background processing, etc.) while W is the maximum "working set" of a single process (which, for large address space processes should be at least 2 MByte.)

On the other hand, Berkeley UNIX currently requires the page tables of all processes to be "locked down", which may be a significant drain for very large address space programs where the data in the address space is in fact infrequently referenced.

## IV. CONCLUSIONS: WHITHER INTERLISP VAX

### A. There aren't any good alternatives

Given the requirements of TECHNOLOGY transfer to universities, industrial and military sites, there are few other options: even though Interlisp-VAX will probably not be cost effective for intensive Lisp users, it may be for those whose requirements are for casual and occasional use of Interlisp or tools developed in it. There are a few alternatives which could benefit from further exploration:

Interlisp-370
There is a version of Interlisp for IBM/370 machines, originally developed at Uppsala University and modified at the Weizmann Institute [Raim]. Interlisp-370 might be a possibility for some sites, although reports from several sources are that the Interlisp-370 is incomplete, and not particularly compatible with other Interlisps, and has serious performance and reliability problems. However, I believe that this alternative should be more seriously explored.

Implementing Interlisp on top of NIL or Franz
This might have been a reasonable way to approach the initial Interlisp-VAX implementation, but it does not seem cost effective at this point.

Emulating Interlisp-D on a VAX
An alternative, not presently explored in any detail, would be to write an Interpreter for Interlisp-D byte codes and run Interlisp-D on a VAX (cf. [Rowan]). Performance would be poor (perhaps a factor of 4-5 slower than currently projected), but code would be more easily transportable.

Automatic conversion of Interlisp programs to other VAX Lisps
This is an approach which has rarely succeeded. Programs which convert between language dialects are heuristic at best, and require considerable hand-holding; for any particular program, converting to another language might be cost effective, but on the whole, it is not.

B. Performance: mixed results

Performance in the Lisp community is often measured in DEC KA-10 or KL-10 equivalents, e.g., "1/4 of the speed as on a KL-10." One would like to be able to draw the inference that, if a KL-10 adequately supports 40 users with 8 actively computing (the rest editing, reading mail, etc.), 1/4 of that would amount to 10 users with 2 actively computing. Unfortunately, these performance figures can be misleading, first because of the wide variation in Interlisp-10 speeds on the same problem, and second because timings on small benchmarks do not give an accurate picture of the number of active users who can be supported in a working environment.

More reasonable estimations of performance can be drawn from experience with VAXes running Franz Lisp or VAXSYMA; while no exact figures are available, experience has been that a VAX 11/780 with 4 MByte real memory can support 30 users, of whom 3 are actively using VAXSYMA. Interlisp working-set and virtual address space requirements will exceed those of VAXSYMA.

Although the VAX is purported to be quite cost effective for FORTRAN, the instruction set is not PARTICULARLY effective for Lisp, and even less so for Interlisp; the "CALLS" instruction, which is intended to be used for function calls in high level languages, assumes a model of the stack which does not match Interlisp's. While the Interlisp-VAX design takes advantage of "CALLS" in a clever way, function call is still relatively more expensive than it is on microcoded machines which can have an Interlisp-specific function call instruction.

Virtual address space and real memory

Although the VAX is a large virtual address space machine, the address space may not be particularly usable on configurations typical in many installations. For example, the following configurations were proposed as "typical" VAX installations:

VAX-11/750 with 2 MByte real memory (maximum for 750)

VAX 11/780 with 4 MByte real memory

VAX 11/780 with 8 MByte real memory (requires additional memory controller)

Also proposed are configurations not currently available: "single-user" VAX machines with memory in the 1-2 MByte range, or 750's and 780's with more memory (requiring 64K RAM chips.)

Because of Interlisp-VAX's large virtual address space and working set, a machine with only 2 MByte of real memory might be able to support at most one or two large address space active users at a time. Generous amounts of disk, swapping space, and real memory will be required -- more so than in Interlisp-10 to support the same users, and much more so than in Interlisp-D or Interlisp-Jericho. Very few time-sharing systems have adequately dealt with giant address spaces for multiple users. The success

of very large-address-space Interlisp-VAX will depend on the cooperation and support of the Berkeley UNIX implementors.

C. There is much left to do

There is an unfortunate tendency to underestimate the magnitude of the task of transporting a system the size and complexity of Interlisp. Interlisp is not merely an interpreter and a few utility routines, but a rich and complex programming environment, with facilities which were heavily influenced by Tenex, its original host operating system. Porting it to another machine and continuing to upgrade it is a major undertaking. I cannot stress this enough.

The publication of the Interlisp Virtual Machine specification [Moore], was an important step forward in the creation of transportable Interlisp, in that it identified a major portion of what the "higher-level" Interlisp support software required in order to run. Unhappily, as complete and well written as that document was, it is not an accurate guide for the construction of a useful Interlisp implementation, in that many areas are designated as being left to the implementor while many Interlisp applications require exact compatibility with Interlisp-10. The VM is also not a good measure of the magnitude of implementing Interlisp. For example, the VM mentions the compiler only in passing; however, providing a reasonable Interlisp compiler is a major portion of the task of transporting Interlisp to a new (non-microcoded) machine.

Transporting Interlisp is harder than merely implementing "some" Lisp dialect. It is much more difficult to be strictly compatible while using the underlying power of the machine to the fullest. Compatibility makes the implementation harder because there is an existing standard against which the implementation can be judged. For a "new" Lisp, it is always possible to declare oneself "done" at almost any point. The necessity of emulating exactly the behavior of another system is what makes the task more difficult.

How much is left to do?

It is difficult to give a "man-month" figure for Interlisp-VAX for several reasons. First, of course, the notion of "man-month" independent of implementor is a well-known paradox: start-up time and personnel training, can delay a project for many months (as in the early months of the Interlisp-VAX project).

Second, there are several tasks ahead which will undoubtedly encounter unforeseen problems. "System shakedown" is a catch-all phrase which can cover many months of discovering problems or previously undetected system requirements. Software completion is not measured well by proportion of lines-of-code written.

Finally, there is a wide range of variation of what is meant by "Interlisp-VAX." On the one hand, an initial version may be available relatively soon. This version will likely have serious performance problems (mainly because of free variable reference and non-tuned code generation),

and will likely be not fully functional or compatible with Interlisp-10. The task of bringing Interlisp-VAX to the level of functionality, performance and reliability of Interlisp-10 and Interlisp-D remains awesome.

Unfortunately, there is not a good perception in the Interlisp user community of the amount of work between the first release and a system which will be acceptable to current Interlisp users; for this reason, the recent "pre-announcement" message [Dyer] was at best misleading for those trying to make plans based on Interlisp-VAX availability. While this initial version might in fact be a reasonable alternative to, say, converting a large Interlisp program to Franz Lisp, (because the conversion cost would be higher than the performance difference would warrant), it will not be comparable with most other Interlisp implementations (-10, -D, -Jericho).

The Interlisp-VAX project is and has been from the beginning drastically undermanned. The initial proposal for implementation of Interlisp-VAX in one year with no existing personnel was at best wishful thinking. Hans Koomen will be leaving within the near future. This is a serious, although possibly not fatal, blow to the continuation of the project, even with the addition of additional staff members (Ray Bates and Don Voreck).

The project needs a team of implementors who are committed to its goals, are qualified to carry it out, and will stick with the project once the initial release has been made: if Interlisp-VAX is to be viable, there needs to be a long-term (3-4 year) commitment to its maintenance and support by a team of qualified personnel. This level of support or greater has been required by every other serious implementation of Lisp that I know of, including Interlisp-10, Interlisp-D, Interlisp-Jericho, and Lisp Machine Lisp. There is no reason why anyone should imagine that Interlisp-VAX would be different.

NOTES

(1)VAX is a trademark of Digital Equipment Corporation.
(2)Unix is a trademark of Bell Laboratories.

BIBLIOGRAPHY

Burton, R.R., et al. "Interlisp-D: Overview and Status." In Papers on Interlisp-D, Xerox Palo Alto Research Center, CIS-5 (SSL-80-4), 1980.
*(Describes the Interlisp-D implementation effort, including some words of wisdom on why implementing Interlisp is hard.)

Dyer, D., et al. INTERLISP-VAX. Message-ID: <[USC-ISIB]17-Jul-81 15:10:10.MILLAR>
*(This was the "official pre-announcement of the availability of Interlisp-VAX.")

Hedrick, Charles. Some Tests of Big Core Images. [Message file] Rutgers University. 30 May 81 0447-EDT.
*(Discusses ELISP implementation on extended address 2060.)

Kashtan, David. UNIX and VMS: Some Performance Comparisons. [Message file] SRI International.
*(Compares performance of VAX/VMS version 1.6 and VM UNIX Berkeley version 2.1.)

Masinter, L. M. and Deutsch, L. P. "Local Optimization in a Compiler for Stack-based Lisp Machines." In Papers on Interlisp-D, Xerox Palo Alto Research Center, CIS-5 (SSL-80-4), 1980.
*(Describes the byte compiler.)

Moore, J. The Interlisp Virtual Machine Specification. Xerox Palo Alto Research Center, CSL 76-5, revised March 1979.

Raim, Martin. Personal communication.

Rowan, William. A Lisp Compiler Producing Compact Code. LISP Conference proceedings, 1980.
*(A byte-code interpreter for compiled MacLisp.)

Teitelman, W. and Masinter, L. The Interlisp Programming Environment. IEEE Computer, April 1981, pp. 25-33.
*(Overview of Interlisp.)

# Appendix B

## AI Handbook Outline

Volumes I and II by Avron Barr and Edward A. Feigenbaum
Volume III by Paul R. Cohen and Edward A. Feigenbaum
Computer Science Department
Stanford University

This is a list of the Chapters in the Handbook. A list of all of the articles in each Chapter follows.

VOLUME I:

I. Introduction
II. Search
III. Knowledge Representation
IV. Understanding Natural Language
V. Understanding Spoken Language

VOLUME II:

VI. Programming Languages for AI Research
VII. Applications-oriented AI Research: Science
VIII. Applications-oriented AI Research: Medicine
IX. Applications-oriented AI Research: Education
X. Automatic Programming

VOLUME III:

XI. Models of Cognition
XII. Automatic Deduction
XIII. Vision
XIV. Learning and Inductive Inference
XV. Planning and Problem Solving

## VOLUME I

I. INTRODUCTION

- A. Artificial Intelligence
- B. The AI Handbook
- C. The AI literature

II. SEARCH

- A. Overview
- B. Problem representation
  - 1. State-space representation
  - 2. Problem-reduction representation
  - 3. Game trees
- C. Search methods
  - 1. Blind state-space search
  - 2. Blind AND/OR graph search
  - 3. Heuristic state-space search
    - a. Basic concepts in heuristic search
    - b. A*--Optimal search for an optimal solution
    - c. Relaxing the optimality requirement
    - d. Bidirectional search
  - 4. Heuristic search of an AND/OR graph
  - 5. Game tree search
    - a. Minimax procedure
    - b. Alpha-beta pruning
    - c. Heuristics in game tree search
- D. Sample search programs
  - 1. Logic Theorist
  - 2. General Problem Solver
  - 3. Gelernter's geometry theorem-proving machine
  - 4. Symbolic integration programs
  - 5. STRIPS
  - 6. ABSTRIPS

III. KNOWLEDGE REPRESENTATION

- A. Overview
- B. Survey of representation techniques
- C. Representation schemes
  - 1. Logic
  - 2. Procedural representations
  - 3. Semantic networks
  - 4. Production systems
  - 5. Direct (analogical) representations
  - 6. Semantic primitives
  - 7. Frames and scripts

IV. UNDERSTANDING NATURAL LANGUAGE

- A. Overview
- B. Machine translation
- C. Grammars
  - 1. Formal grammars
  - 2. Transformational grammars
  - 3. Systemic grammar
  - 4. Case grammars
- D. Parsing
  - 1. Overview of parsing techniques
  - 2. Augmented transition networks
  - 3. The General Syntactic Processor
- E. Text generation
- F. Natural language processing systems
  - 1. Early natural language systems
  - 2. Wilks's machine translation system
  - 3. LUNAR
  - 4. SHRDLU
  - 5. MARGIE
  - 6. SAM and PAM
  - 7. LIFER

V. UNDERSTANDING SPOKEN LANGUAGE

- A. Overview
- B. Systems architecture
- C. The ARPA SUR projects
  - 1. HEARSAY
  - 2. HARPY
  - 3. HWIM
  - 4. The SRI/SDC speech systems

VOLUME II

VI. PROGRAMMING LANGUAGES FOR AI RESEARCH

- A. Overview
- B. LISP
- C. AI programming-language features
  - 1. Overview
  - 2. Data structures
  - 3. Control structures
  - 4. Pattern matching
  - 5. Programming environment
- D. Dependencies and assumptions

VII. APPLICATIONS-ORIENTED AI RESEARCH: SCIENCE

- A. Overview
- B. TEIRESIAS
- C. Applications in chemistry
  - 1. Chemical analysis
  - 2. The DENDRAL programs
    - a. Heuristic DENDRAL
    - b. CONGEN and its extensions
    - c. Meta-DENDRAL
  - 3. CRYSALIS
  - 4. Applications in organic synthesis
- D. Other scientific applications
  - 1. MACSYMA
  - 2. The SRI Computer-based consultant
  - 3. PROSPECTOR
  - 4. Artificial Intelligence in database management

VIII. APPLICATIONS-ORIENTED AI RESEARCH: MEDICINE

- A. Overview
- B. Medical systems
  - 1. MYCIN
  - 2. CASNET
  - 3. INTERNIST
  - 4. Present Illness Program
  - 5. Digitalis Therapy Advisor
  - 6. IRIS
  - 7. EXPERT

IX. APPLICATIONS-ORIENTED AI RESEARCH: EDUCATION

- A. Overview
- B. ICAI systems design
- C. Intelligent CAI systems
  - 1. SCHOLAR
  - 2. WHY
  - 3. SOPHIE
  - 4. WEST
  - 5. WUMPUS
  - 6. GUIDON
  - 7. BUGGY
  - 8. EXCHECK
- D. Other applications of AI to education

X. AUTOMATIC PROGRAMMING

- A. Overview
- B. Methods of program specification
- C. Basic approaches

D. Automatic programming systems
   1. PSI and CHI
   2. SAFE
   3. The Programmer's Apprentice
   4. PECOS
   5. DEDALUS
   6. Protosystem-1
   7. NLPQ
   8. LIBRA

VOLUME III

XI. MODELS OF COGNITION

A. Overview
B. General Problem Solver
C. Opportunistic problem solving
D. EPAM
E. Semantic network models of memory
   1. Quillian's semantic memory system
   2. HAM
   3. ACT
   4. MEMOD
F. Belief systems

XII. AUTOMATIC DEDUCTION

A. Overview
B. The resolution rule of inference
C. Nonresolution theorem proving
D. The Boyer-Moore theorem prover
E. Nonmonotonic logic
F. Logic programming

XIII. VISION

A. Overview
B. Blocks-world understanding
   1. Roberts
   2. Guzman
   3. Falk
   4. Huffman-Clowes
   5. Waltz
   6. Shirai
   7. Mackworth
   8. Kanade
C. Early processing of visual data
   1. Visual input
   2. Color

3. Preprocessing
4. Edge detection and line finding
5. Region analysis
6. Texture

D. Representation of scene characteristics
1. Intrinsic images
2. Motion
3. Stereo vision
4. Range finders
5. Shape-from methods
6. Three-dimensional shape description and recognition

E. Algorithms for vision
1. Pyramids and quad trees
2. Template matching
3. Linguistic methods for computer vision
4. Relaxation algorithms

F. Vision systems
1. Robotic vision
2. Organization and control of vision systems
3. ACRONYM

XIV. LEARNING AND INDUCTIVE INFERENCE

A. Overview
B. Rote learning
1. Issues
2. Rote learning in Samuel's Checkers Player

C. Learning by taking advice
1. Issues
2. Mostow's operationalizer

D. Learning from examples
1. Issues
2. Learning in control and pattern-recognition systems
3. Learning single concepts
a. Version space
b. Data-driven rule-space operators
C. Concept learning by generating and testing plausible hypotheses
d. Schema instantiation
4. Learning multiple concepts
a. AQ11
b. Meta-DENDRAL
c. AM
5. Learning to perform multiple-step tasks
a. Samuel's Checkers Player
b. Waterman's Poker Player
c. HACKER
d. LEX
e. Grammatical inference

XV. PLANNING AND PROBLEM SOLVING

A. Overview
B. STRIPS and ABSTRIPS
C. Nonhierarchical planning
D. Hierarchical planners
   1. NOAH
   2. MOLGEN
E. Refinement of skeletal plans